Cache Digests*

Alex Rousskov
Duane Wessels
National Laboratory for Applied Network Research

April 17, 1998

Abstract

This paper presents Cache Digest, a novel protocol and optimization technique for cooperative Web caching. Cache Digest allows proxies to make information about their cache contents available to peers in a compact form. A peer uses digests to identify neighbors that are likely to have a given document. Cache Digest is a promising alternative to traditional per-request query/reply schemes such as ICP.

We discuss the design ideas behind Cache Digest and its implementation in the Squid proxy cache. The performance of Cache Digest is compared to ICP using real-world Web caches operated by NLANR. Our analysis shows that Cache Digest outperforms ICP in several categories. Finally, we outline improvements to the techniques we are currently working on.

1 Introduction

One of the most difficult problems in the design of Web cache hierarchies is efficiently locating objects held in neighbor caches. When a cache needs to forward a request, how does it know whether to use a sibling, a parent, or perhaps the origin server directly? Many caches in operation today utilize the Internet Cache Protocol (ICP)[1, 2] for this purpose.

Although ICP works reasonably well, it can add significant delays to cache misses. When neighbor caches are located close to each other (e.g. within an organizational LAN), ICP delays are usually not significant. In a wide-area environment, however, ICP becomes troublesome.

This paper proposes an alternative to ICP, which we call *Cache Digests*. A cache digest, based on Bloom Filters, is essentially a lossy compression of all cache keys with a lookup capability. Digests are made available via HTTP, and a cache downloads its neighbor's digests at startup. By checking a neighbor's digest, a cache knows (with some uncertainty) whether or not that neighbor holds a given object.

Of course, nothing comes for free, and Cache Digests have some drawbacks which must be taken into consideration. Bloom Filters do not perfectly represent all of the items they encode. Occasionally the Bloom Filter will incorrectly report some item is present, when in fact it is not. For Web caches, this means we generate a remote cache miss when we were expecting a cache hit. Another penalty comes from the additional memory required to store cache digests.

*This work is supported by grants of the National Science Foundation (NCR-9521745, NCR-9616602).

In this paper we describe our design and implementation of Cache Digests in Squid 1.2, and how we measure their effectiveness. We ran Squid with cache digest support on the NLANR caches for a period of six days. A special modification was made such that Squid used Cache Digests for half of the cache misses, and ICP for the other half. During the study period, we collected measurements on service times, network traffic, hit/miss ratios, and memory usage.

Our results show that Cache Digests successfully eliminate the ICP delays, at the expense of some false misses. For our study, Cache Digests consumed lower network bandwidth overall, but with much more bursty traffic compared to ICP. Memory usage increases because Cache Digests require additional memory, but ICP does not.

2 The Problem

Connecting a Web cache into a hierarchy or mesh offers additional benefits from Web caching technology. To determine which member of a cache group (if any) a cache miss should be forwarded to, many caches use the lightweight Internet Cache Protocol (ICP)[1, 2]. ICP messages are transmitted via UDP. A message consists of a 20 byte fixed-format header plus a URL. A cache sends an ICP_QUERY message to one or more neighbor caches. The neighbors reply with ICP_HIT or ICP_MISS messages to indicate the presence or absence of the named object in their cache.

The most significant disadvantage of ICP is the additional delay introduced by the query/reply exchange. Because caches can process ICP queries very quickly, the delay is a little more than the network round-trip time (RTT) between a pair of caches. Obviously the delay penalty caused by ICP will depend upon the cache's proximity to each other. For LAN-connected caches, ICP delays should be acceptable, but not necessarily for WAN caches. Figure 1 shows the ICP delays experienced by the NLANR caches.

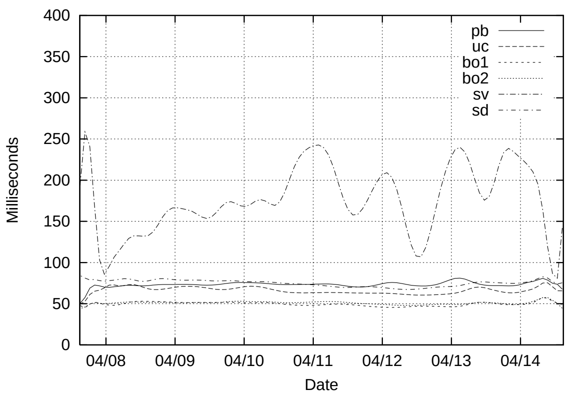


Figure 1: ICP delays experienced by the NLANR caches for a week in April, 1998. These values are the five-minute medians of the amount of time elapsed between sending out ICP queries and choosing the next-hop cache. The five flat curves are from the vBNS-connected caches where the network round-trip time between nodes ranges from 40–60 milliseconds. The other curve (SV) is from our cache at FIX-West. This cache only uses ICP with overseas caches in Asia, Australia, and New Zealand. NLANR operates other caches (PA, SJ) which are not shown on this plot because they do not utilize ICP at all.

Another disadvantage is that ICP's effectiveness drops off noticeably when sending more than four queries per cache miss. ICP scales poorly because the number of network messages is proportional to the product of the number of neighbors and the number of HTTP requests.

ICP does provide a couple of important advantages as well. Because of its UDP transport, ICP replies (or lack of them) inherently indicate current network conditions. If a network path is congested, ICP replies will be delayed longer than normal, or perhaps dropped entirely. Similarly, receipt of an ICP reply indicates the neighbor cache is up, running, and serving requests.

Another advantage of ICP is its SRC_RTT feature. When this feature is enabled, ICP replies will include an ICMP-based measurement of the network RTT between the neighbor cache and the origin server (if available). This provides a logical tie-breaking mechanism if all neighbors return ICP_MISS. The request can be forwarded to the neighbor which is closest (i.e. lowest RTT) to the origin server.

In the next section we present an alternative method of object location which eliminates the query/reply step and its associated delays.

3 Cache Digests

Instead of a query/reply scheme, we are in search of a technique which allows caches to efficiently inform each other about their contents without any per-request delays. One approach might be to transfer the entire list of cache keys (i.e. URLs) from one cache to another. For a cache holding 1 million objects, and an average URL length of 50 bytes[1], this results in a data transfer of approximately 50 MBytes. We might, instead, use MD5 hashes of the URLs, but this still results in a transfer of 16 Mbytes.

To reduce the "cache directory" size even further, we might select a subset of the standard 128 MD5 bits, at the expense of some accuracy. Using only 8 of the MD5 bits reduces our directory size to a more reasonable 1 MB. But then we have only 256 possible hash values, and the number of collisions is unacceptably large. However, there is still hope for a small directory size with relatively low occurrence of collisions. Bloom Filters[3] have been been well-established in the field of databases for many years, and offer small directory sizes with low collision probabilities.

3.1 Bloom Filters

A Bloom Filter is an array of bits, some of which are on and some of which are off. To add an entry to the bloom filter, a small number of independent hash functions are computed for the entry's key (e.g. URL). The hash function values specify which bits of the filter should be turned on. To check whether a specific entry is in the filter, we calculate the same hash function values for its key and examine the corresponding bits. If one or more of the bits is off, then the entry is not in the filter. If all bits are on, there is some probability that the entry is in the filter.

The size of a Bloom Filter determines the probability an "all-bits-on" lookup is correct. A smaller filter size will result in more errors than a larger one for the same data. We use the terms *hit* and *miss* to indicate whether or not the bits of the Bloom Filter predict that a given object is in the filter. Furthermore, the terms *true* and *false* describe the correctness of the prediction. Thus, we have:

***true hit*:** The filter correctly predicts an entry is in the cache.

***false hit*:** The filter incorrectly predicts the entry is in the cache.

***true miss*:** The filter correctly predicts the entry is not in the cache.

[1] The average URL length calculated from `ftp://ircache.nlanr.net/Traces/sv.sanitized-access.980413.gz` is 48.57 bytes.

false miss: The filter incorrectly predicts the entry is not in the cache.

The distribution of these numbers will indicate the effectiveness of our algorithms. Obviously, we want to maximize the *true* values (thus minimizing the *false* ones). By its very nature, a Bloom Filter will always have a non-zero number of false hits. This is the price paid for its compact representation. When the Bloom Filter is perfectly synchronized with its source, there will be zero false misses. In our implementation, cache digests are not always perfectly synchronized, so we should expect some number of false misses.

Our Cache Digest design has several layers which are further described below.

3.2 Building the Bloom Filter

What are the tradeoffs between cache digest size and effectiveness? False miss probabilities can be calculated, but do they hold in practice? When we use more bits per object, the percentage of false hits should decrease. A related question is the optimal density of the Bloom Filter. As objects are added to the filter, its density will increase.

Should all cached objects be treated equally in the digest? We might be able to decrease the cache digest size (or decrease the false hit ratio) by allocating fewer bits to objects which are less likely to be requested. The penalty for using a variable number of hash functions is a higher false miss probability. Note that the peer that checks our cache digest will always use the same (maximum) number of hash functions.

For example, when hashing a fresh object that has been accessed at least twice, we can use four hash functions. On the other hand, for stale objects that were accessed only once, we could use only two hash functions. The number of possible optimizations is virtually unlimited because many object properties can be used for determining the likeliness of future requests to the object. However, it is not yet clear how effective those optimizations will be.

3.3 Local Storage

Cache Digests are relatively large data structures (e.g. 200 KB to 2 MB). It may be tempting to keep local digests entirely on disk. A proxy generates its cache digest only for the benefit of its neighbors. If digests are kept only on disk, they must be rebuilt from scratch to be updated.

Rebuilding a digest may be CPU intensive. Thus, we have a tradeoff between memory utilization and CPU overhead. To make things even more interesting, note that rebuilding a digest also requires allocating temporary memory for storing the new digest while it is being built. Consequently, one has to choose between having periodic peaks in memory and CPU usage and constant lower memory utilization.

Finally, if deleting from the digest is not supported, even a memory-resident digest must be rebuilt from scratch on a periodic basis to erase stale bits and prevent digest pollution (see next subsection).

3.4 Local Updates and Deletes

There are two kinds of updates to a local digest. First, new entries need to be added as they enter the cache. Second, if an object gets purged from the cache, it is desirable to undo its effect on the local digest.

Adding new entries to a local digest is simple. If these updates will be propagated to neighbors, the application must keep a history of recent updates.

It is not possible to delete entries from a standard bloom filter. Any bit in the array which is on may have been set due to any number of entries. In other words, two or more unique keys may turn on the same bit. One way to support deletes is to use integer counters instead of single bits. This would necessarily increase the digest size, and we might need to check for overflows. The update history should also include deletions if they are supported.

3.5 Digest Dissemination and Update Propagation

Which transport protocol is best for exchanging digests? ICP is fairly well established as a lightweight alternative to HTTP for cache-to-cache communications. If Cache Digests are transmitted as ICP messages (via UDP), the design must either be able to tolerate gaps from lost packets, or implement a message retransmission scheme. If reliability is important, TCP would make a better choice. For TCP we might either develop a customized digest exchange protocol, or exchange digests via HTTP.

In exchanging cache digests, should we employ a *push* or *pull* technique? The push model puts the server in control of distributing updates to its clients. We might like to use push because the server knows exactly how rapidly its cache contents changes. However, this also requires that the server know which of its clients are digest-aware caches, and which are browsers, or caches which do not support digests. Additionally, the digest-aware client must be able to receive some data which it did not specifically request.

Once a digest has been given to a client cache, how should we update it? With no updates, the digest slowly becomes stale, and will have increasing numbers of false hits and false misses. Is there an efficient way to incrementally update cache digests? Or do we need to send the entire digest every time?

3.6 Remote Storage

A proxy must keep peer digests in memory because they are consulted on most misses. It also may make sense to store copies of digests on disk. When a proxy gets restarted, disk resident digests can be reused if they are fresh enough. Note that expiration and other useful information is stored on disk along with the digest itself.

Another possible use of disk resident digests is sharing those digests with other proxies. It certainly makes sense to fetch a cached digest from a neighbor rather than from its original producer, if the neighbor is closer. By storing digests on disk, we can treat them as any other cached object. This greatly simplifies the implementation.

3.7 Handling False Hits

How are false hits to be handled? For sibling relationships, we cannot tolerate a significant (or perhaps any) number of false hits. This violates the definition of a sibling relationship. We will require some feature in HTTP to detect false hits and deal with them appropriately.

Design decisions on each layer can be done independently. However, only a balanced architecture will result in good performance and scalability. The following section describes motivation behind our design choices.

4 Proposed Approach

Balancing design alternatives on all layers is a fine art. Our goal was to find a simple yet robust combination rather than a perfect optimal solution. Our choices often leave a lot of freedom for further optimization and tuning.

4.1 Building a Digest

Cache digests are built using a fixed set of hash functions: hash values are simply 4 byte sections of a standard 16 byte MD5 computed over a URL string. The hash values, modulo the digest size, determine which bits will be turned on. Each cache determines the size of its digest independently from its peers. The same holds for the number of bits per object. The size of a digest is likely to depend on the current cache size and maximum cache capacity.

Note that the proposed scheme allows for a very efficient lookup implementation on foreign digests while allowing great flexibility in building a digest. For example, to reduce the number of false hits, a proxy may decide to apply fewer hash functions for old or stale entries. This will affect the way a cache digest is computed. However, the lookup algorithm for peers remains unchanged. In general, Cache Digest separates digest computation from digest use.

4.2 Storing a Digest

A proxy maintains both memory and a disk resident copies of its own digest. The in-memory copy is used for fast updates. If, for some reason, the memory copy is unavailable, requests for the digest can still be satisfied with the on-disk copy. In fact, keeping a memory resident copy is actually not required. A proxy may decide to recompute its digest from scratch instead of constantly updating the in-memory copy. When a digest is written to disk, it is treated as any other cached object. A proxy does not have to employ special algorithms to share its local digest with its peers. If a peer is allowed to retrieve cached documents from a proxy, it can retrieve a cache digest as well. Clearly, more complicated access controls can be implemented, but they are not addressed here[2].

The local digest can be recomputed at configurable (but fixed) time intervals. A smarter algorithm could monitor the density of the current digest and trigger rebuild if, for example, the number of bits turned on exceeds some threshold. A trigger based on the cache's object expiration rate is another alternative.

The presence and quality of local update algorithms is transparent to other proxies. A proxy must, however, attach an expiration date to its cache digest so the peers will know when the digest may become out-of-date. A precise expiry time is not required, and peers are not required to discard expired digests.

[2]However, we should note that we are aware of some potential security issues. Given a list of URLs and a cache digest, someone can discover (via brute-force methods) which objects are likely present in a cache and which are not.

4.3 Disseminating Digests

We propose to use a *pull* technique for disseminating cache digests. Pull fits very well with the current distribution and access control schemes for Web objects. Push requires a parent proxy to maintain state data for all of its children. Moreover, a child cache may not be willing to accept a new digest at an arbitrary time selected by a parent proxy[3].

A similar problem (i.e., proxy is not willing to serve a digest to a child cache) does exist with the pull approach. However, techniques for resolving such conflicts are already well developed and do not waste bandwidth on transmitting potentially huge PUT requests, only to receive a negative reply.

To preserve bandwidth and proxy resources, peers will use an "If-Modified-Since" request or equivalent technique when refreshing neighbor digests. Moreover, it may be desirable to fetch the digest of a remote parent via a local neighbor. Remember that digests are treated as any other Web object and can be disseminated using existing cache hierarchies[4].

4.4 Digest Updates

After a proxy gets a digest from its neighbor, the digest is no longer synchronized with the contents of the neighbor's cache. This difference may be acceptable, if the following two conditions hold until the next digest transfer: (1) the neighbor does not add a lot of new, popular objects to its cache, and (2) only unpopular documents are purged from the neighbor's cache. If both conditions hold, the inconsistency embedded within an unsynchronized digest is harmless because it does not affect documents that are actually being requested by clients.

If at least one of the conditions is false, it may be desirable for a proxy to notify its digest users about recent changes in its cache contents. It is not obvious how frequently such notifications should be sent. Very frequent updates will reduce the number of false hits and misses. However, they may require significant amounts of bandwidth.

If updates are to be supported, we propose to "piggyback" update messages in HTTP replies by using two custom HTTP headers: `X-Accept-Digest-Update` and `X-Digest-Update`. The first header is used in the requests to notify the recipient that originator can accept digest updates in the reply for that particular message. The second header is used in the reply to send current updates.

The proposed piggybacking scheme has several major implications:

1. We do not require any new protocols to handle exceptional conditions; HTTP takes care of that.
2. No extra messages are generated between cooperating proxies.
3. Parents are not required to maintain state information about child caches.
4. There are practical limits on the amount of information we can or should place into HTTP headers. However, we do not envision this as a real problem. We do not want to significantly increase the overall message size. Update headers should be limited to some fraction (e.g. 10%) of the message content-length, with an upper limit of 2 KB or so.

[3] Most bandwidth charging schemes are "receiver-pays."

[4] Another potential security problem exists if cache digest objects become corrupted or sabotaged.

Note, it may be the case that only one of the update conditions described above holds true. Common sense suggests that (2) (only purging unpopular objects) is the most likely to hold. In this case, we do not need to support delete notifications, only addition notification would be sent. Supporting deletes is simple to implement, but does consume extra memory on the proxy which maintains the digest[4].

4.5 Digest Accuracy

Cache digests increase the uncertainty level in cache requests. False misses lead to lower hit ratios for peers and reduce hierarchy utilization. By design, false misses should rarely occur[5], whereas false hits are much more likely. We propose a relatively simple mechanism to handle false hits.

When a proxy requests an object based on the cache digest hit prediction, and the request is sent to a sibling (which cannot forward misses for us), we add a standard HTTP cache control directive called `only-if-cached`. If the request turns out to be a miss, according to HTTP/1.1[5], the proxy will send a 504 (Gateway Timeout) reply. Upon receipt of a 504 reply, the originating proxy will either forward the request to a parent or directly to the origin server. When such a request is sent to a parent (which can forward misses), the `only-if-cached` directive is not added.

Note that other resource discovery protocols such as ICP and HTCP[6] are facing the same problem with uncertain information. By the time an ICP or HTCP reply comes back, and the subsequent HTTP request reaches the peer, the object may get purged, thereby resulting in a false hit. The probability of a false hit is lower for these protocols, however, the solution proposed above will work for ICP and HTCP as well as for Cache Digests.

5 Implementation and Methodology

We have built Cache Digest support into Squid version 1.2.beta20. At startup, each cache builds a digest of its own contents. The digest is built as the *swap.state* file is read, and continually updated as new objects enter the cache. Because deletes are not supported, the entire cache digest is rebuilt periodically (every hour).

We choose to transfer cache digests as HTTP messages. Digests may be quite large, and individual UDP messages are limited to a system's socket buffer size (typically ranging from 9–64 KB). Additionally, there are numerous commercial cache products available which do not support ICP. Using HTTP increases the chance that Cache Digests will someday be implemented these products.

The peer cache's digests ("peer digests") are requested on demand. That is, Squid will not request a peer's digest until it is needed. Cache digests are served as standard HTTP replies, with an Expires header based on the digest rebuild period. Thus, we know when to request a fresh digest from a peer. Cache digest replies may be swapped to disk (as standard cache objects). A cache may serve its own digest to others from the on-disk copy. A cache may need to swap in a peer's digest if the peer returns an HTTP 304 (Not Modified) reply in response to an update request.

At present, we do not properly handle false hits. For a parent relationships, this is not a problem because a proxy is allowed to cause a cache miss at its parent. For sibling relationships, on the other hand, false hits need to be dealt with properly. As discussed above (section 4.5), we intend to use the `only-if-cached` directive. Our problem is that Squid is not fully programmed to properly

[5] Recall that an up-to-date cache digest produces no false misses.

handle a 504 reply and re-forward the request to a parent cache or directly to the origin server.

With ICP, there are a couple of different tie-breaking mechanisms available. Normally we use the peer advertising the lowest RTT to the origin server (the SRC_RTT feature), or the first neighbor to reply. For Cache Digests, we select the neighbor which is closest to *us*. We would prefer to select the peer which is closest to the *origin server*, but currently the only way Squid receives peer RTT measurements is via ICP[6]. If there are not any RTT measurements available, we employ a round-robin approach.

If all peer digests report a miss, where should we forward the request? Again, with ICP's SRC_RTT feature we can choose the parent which is closest to the origin server, or go directly to the origin server if our cache is the closest. Our present cache digest implementation always forwards directly to the origin if all peer digests miss.

Because we need to measure the effectiveness of our cache digest implementation, we must categorize each request as a true hit, true miss, false hit, or false miss. We calculate these values for the cache as a whole, and for each configured neighbor which also supports Cache Digests.

- We know we have a *true hit* when the peer digest indicates the object is present, and the HTTP `X-Cache` reply header[7] specifies a hit.
- We know we have a *true miss* when the peer digest indicates the object is not present, and the `X-Cache` header specifies a miss. Recall, above we stated requests are never forwarded to a neighbor if its digest indicates a miss, so counting true misses is not usually possible. In section 6.1 we describe how we count false misses for this study.
- We know we have a *false hit* when the peer digest indicates the object is present, but the `X-Cache` header specifies a miss.
- We know we have a *false miss* when the peer digest indicates the object is not present, but the `X-Cache` header specifies a hit. Again, counting false misses is not possible in general. Section 6.1 describes our approach for this study.

At the present time, we do not update cache digests with HTTP headers. In fact, we have not yet seen sufficient, real evidence that such updates would make a significant impact in digest accuracy.

6 Framework and Results

6.1 Squid Modifications and Instrumentation

In addition to the support for Cache Digests described above, we also modified Squid in specific ways for this study. The most important modification allows us to compare the performance of Cache Digests to ICP on a single machine. Ideally, we could have two machines at each location receiving similar request streams, one using ICP and the other Cache Digests. Instead, we modified Squid to give each machine a "dual personality." A randomly generated number determines if Squid

[6]This is not a requirement, however, and in the future we may implement a bulk exchange of network RTT measurements.

[7]`X-Cache` is not a standard HTTP reply header. It is an extension header used by Squid to indicate if the request was a *hit* or a *miss*. Every Squid cache adds its own `X-Cache` header.

will use ICP or Cache Digests for a given request (local cache misses only of course). We split the balance evenly, so half of the time we use ICP and the other half Cache Digests.

This technique also allows us to count true and false misses. When Squid selects ICP, we also check our peer's cache digests to find out what they predict about the request. If the cache digests predict a miss, but the `X-Cache` header indicates a hit, then we count a *false miss*. Conversely if both the cache digests and `X-Cache` header indicate a miss, we have a *true miss*.

We have also instrumented Squid to keep client- and server-side service time histograms for both ICP and Cache Digest modes. The client-side service time is the time elapsed between accepting and closing the client HTTP connection[8]. The server-side service time is the time elapsed between beginning the neighbor selection process (i.e. it includes the ICP query/reply phase) and reading the last byte of the server's reply.

Squid 1.2 already counts the number and size of ICP messages sent and received to/from the network. We added similar counters for the Cache Digest network traffic.

One of the tradeoffs for Cache Digests is increased memory usage. We count the number of bytes required for storing local and remote cache digests. ICP does not utilize any additional process memory.

6.2 The NLANR Caches

Our modified version of Squid 1.2 has been running since April 10, 1998 on the NLANR caches[7]. We collect statistics from each cache at five minute intervals. This data is presented in the following sections. To understand some of the data, it is helpful to also understand how the NLANR caches are configured, especially their peering arrangements.

There are currently eight NLANR caches, whose locations and configurations are summarized in the table below:

Name	City	Network or Exchange Point	RAM, MB	Cache Size, GB
PB	Pittsburgh, PA	vBNS[9]	512	16
UC	Urbana-Champaign, IL	vBNS	512	16
BO1	Boulder, CO	vBNS	512	16
BO2	Boulder, CO	vBNS	512	21
SV	Silicon Valley, CA	FIX-West	512	16
SD	San Diego, CA	vBNS	256	8.8
PA	Palo Alto, CA	Digital's PAIX	512	26
SJ	San Jose, CA	MAE-West	512	16

The PA and SJ caches do not have any neighbors, so they are not included in this analysis. However, PA does have international cache clients helping us test our Cache Digests implementation. The remaining caches each receive 0.4–1.2 million requests per day, serving 5–14 GB to 125–200 client caches throughout the world.

The following summarizes the NLANR cache peering configurations:

- PB, UC, BO1, BO2, and SD form the "vBNS club." These caches will forward requests to each other for all *com*, *net*, *org*, *edu*, *gov*, *mil*, and *us* domains. In the case of ICP, this means

[8]For persistent connections it is the time between reading the first request byte and writing the last reply byte.
[9]The nationwide OC-12 backbone operated by MCI for the National Science Foundation. `http://www.vbns.net/`

Proxy	Service Time, msec	
	ICP	Digest
PB	538	371
UC	608	420
BO1	506	394
BO2	537	371
SV	1633	1200
SD	732	505

Table 1: Median client service times for the NLANR caches. The data was collected during the first 48 hour period. Cache Digest consistently outperforms ICP by a large margin. Note that the difference in response time is significantly larger than one RTT between these caches (approximately 40-60 msec). ICP often has to wait for more than one reply before selecting a peer.

one of these caches sends an ICP query to every other (four total) for each local cache miss. Note, 60% of all requests we receive are for an origin server in the *com* domain.

- SV has a few international cache peers (Korea, New Zealand, Australia, Taiwan). SV does not peer with the other NLANR caches, with the exception of BO for *th* domains.
- SD has a few international cache peers (South Africa, Mexico, Brazil, Russia).
- PB has a few international cache peers (Sweden, Netherlands, United Kingdom). PB also receives requests from the other "vBNS club" caches for many European domains.
- We use *cache_host_acl* configuration lines to ensure that any given request passes through no more than two NLANR caches.

6.3 Overview of Results

In the following sections, we compare the differences between ICP- and Cache Digest-based neighbor selection. We are interested in three parameters: service times, digest accuracy, and network traffic. For these parameters, we present tables of values for every cache. For many of the values we also include plots from the PB cache. Our data has been gathered over a six day period (April 10–15, 1998).

6.4 Service Times

Table 1 lists the median client response time for the first 48 hour period of our experiment. On all caches, Cache Digests perform noticeably better than ICP. Cache Digests eliminate ICP query delays for local misses. Another advantage of Cache Digests is that increasing the number of peers will not slow down client requests. ICP, on the other hand, cannot scale well with the number of clients.

Figure 2 shows the cumulative distribution of client-side service times for a 24-hour period on PB. This includes only cache misses on PB which also used either ICP or Cache Digests to select the next-hop cache[10]. The plot on the left shows that for 85% of such requests, Cache Digests provide

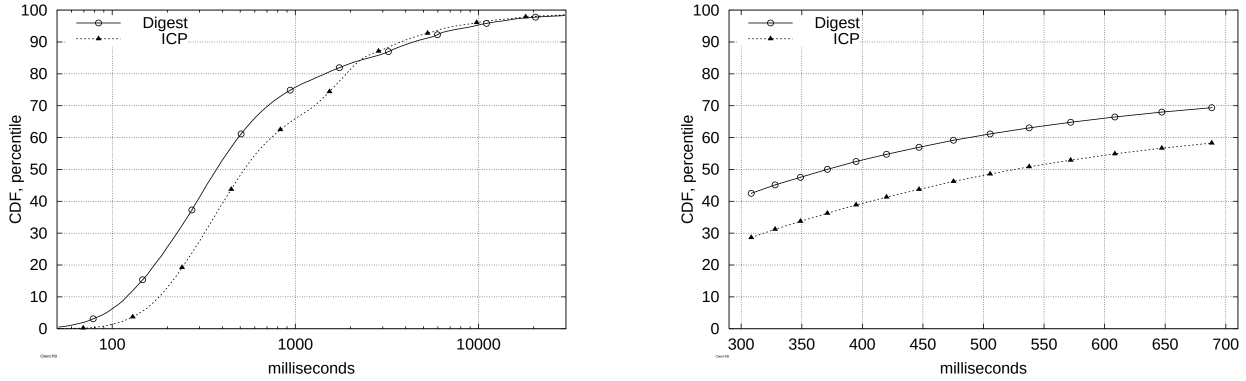


Figure 2: Client-side service times for the PB cache. On the left is a logarithmic plot which clearly shows that Cache Digests provide lower service times to cache clients. A closeup on the right, with a linear X-axis, shows the Cache Digest median is 371 msec, while the ICP median is 538 msec.

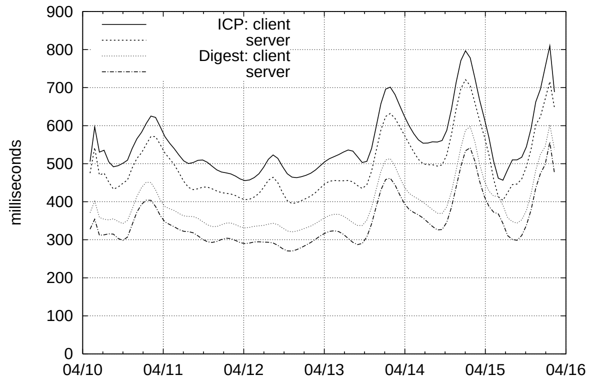


Figure 3: Median client- and server-side service time values for PB during the duration of this study. These median values for 5-minute intervals are smoothed by gnuplot's "smooth bezier" function. The elimination of ICP delays is clearly evident. Cache digests always perform better than ICP-based selection.

significantly lower service times. The remaining 15% have nearly identical service times, perhaps indicating that the client connection is a bottleneck.

The right side of Figure 2 shows a close-up of the area where both curves cross the median. Here, with a linear X-scale, the improvement is easier to see. The Cache Digest median is 69% of the ICP-based value (371 ms vs. 538 ms).

In Figure 3 we plot PB's client- and server-side median service times for the entire 6 day period of our study. This graph shows that cache digests consistently perform better than ICP at all times of the day. Both show increases during peak times, when the Internet becomes congested. Qualitatively, it appears that congestion affects ICP a little bit more than Cache Digests.

6.5 Digest Accuracy

Table 2 is essential in analyzing digest accuracy. Smaller digests result in higher false hit ratios. Stale digests may generate too many false misses. In this table, the number of false misses is negligible on all servers. However, we are disappointed with the high false hit percentages and intend to investigate these further. We might not actually have 18% false hits, but rather, we may not be properly counting them. Counting is somewhat tricky because we must take into

[10] We always forward *cgi-bin* and query requests directly to origin servers.

Proxy	Hit		Miss		Totals			
	true	false	true	false	true	false	hit	miss
pb	33.6	18.1	48.0	0.3	81.7	18.3	51.7	48.3
uc	34.7	15.5	49.4	0.3	84.1	15.9	50.2	49.8
bo1	42.2	17.3	40.1	0.4	82.3	17.7	59.5	40.5
bo2	38.0	17.3	44.3	0.5	82.3	17.7	55.2	44.8
sv	25.7	6.6	67.7	0.0	93.4	6.6	32.4	67.7
sd	39.4	11.9	48.5	0.2	87.9	12.1	51.3	48.7

Table 2: Digest accuracy ratios (%). All caches have very low false miss ratios, which may indicate that most new documents entering a cache are not requested again soon. Thus, delete notification is not required to keep digest hit ratios constant. Our false hit ratios are relatively high. We are working on identifying and solving this problem. The total true ratio needs to be increased to at least 95% before Cache Digests can become attractive for practical use.

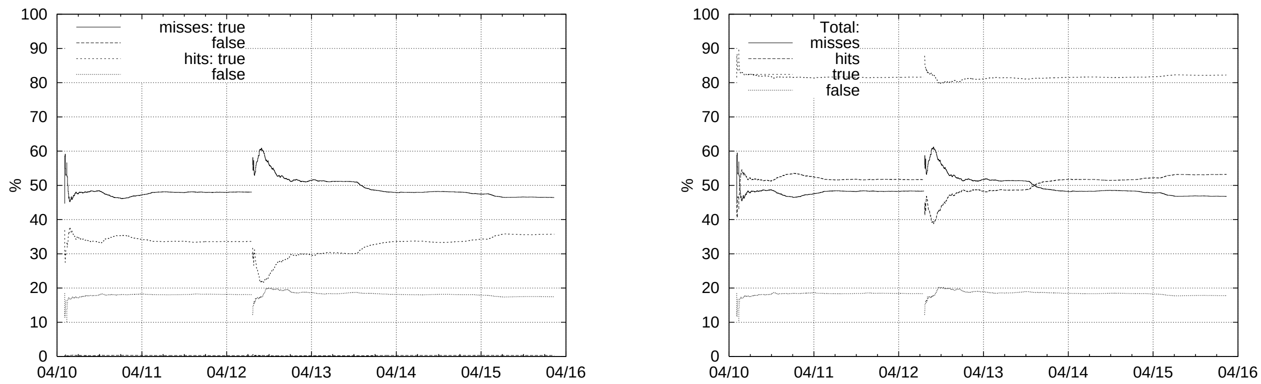


Figure 4: On the left, percentages of true hits, false hits, true misses, and false misses for PB. Note that the percentages are relatively unchanging, except at cache restarts. On the right, percentages of hits, misses, true guesses, and false guesses. Hits and misses are complimentary (together they add to 100%), as are the true and false guesses.

consideration things such as `Pragma: no-cache` and certain `Cache-Control` headers. Also, Caches with differing refresh parameters will also produce remote misses when hits are expected.

In Figure 4, we plot the relative percentages of true hits, false hits, true misses, and false misses for PB. Note that the percentages are quite consistent, except at cache restarts. On the left graph, the top curve (true misses, at about 50%) indicates the number of times peer digests accurately specify they do not contain the requested object. The next curve (true hits, at about 32%) indicates how often at least one of the peer digests accurately specifies it does contain the object. Next, false hits, at about 18%, indicates how often at least one of the peer digests incorrectly specifies it contains the object. The final curve, false misses, is very near to zero, and not visible on this plot.

On the right, we sum the percentages both ways and plot their values over time. The Cache Digest predictions are true about 81% of the time. Conversely, they are false 19% of the time. The hit and miss ratios are nearly equal to each other, just above and below 50%.

Proxy	Traffic rate, KB/hour			
	ICP		Digest	
	Sent	Received	Sent	Received
PB	5.4	5.4	3.2	2.7
UC	7.1	7.1	4.2	2.6
BO1	5.3	5.3	3.7	2.6
BO2	6.1	6.1	5.5	4.7
SV	1.1	1.1	4.8	1.2
SD	7.2	7.2	2.3	2.8

Table 3: Transfer rates for ICP and Cache Digests. Values for ICP traffic are calculated as if we are not using Cache Digests. Unfortunately, our approximation looses some accuracy. In practice, incoming and outgoing ICP bandwidth differs very slightly. Cache Digests have smaller transfer rates in both directions compared to ICP. However, further analysis shows that Cache Digests traffic is bursty unless special optimizations are applied (see discussion of Figure 5).

6.6 Network Traffic

Tables 3 and 4 summarize incoming and outgoing traffic for the NLANR caches. ICP has higher traffic volume and, of course, generates many more messages compared to Cache Digests. Differences in Cache Digests figures among proxies are due to different cache sizes and access patterns. When a small proxy is cooperating with a large one, the larger proxy sends more information than it can receive back. In practice, this may lead to interesting inter-proxy agreements to compensate for an "unfair trade."

The problem of asymmetric traffic is not specific to Cache Digests though. Moreover, with cache digests, proxies may have better control on the amount of data transmitted within the hierarchy. For example, a large proxy that wants to limit the number of requests from a small peer may give the latter a cache digest with fewer bits turned on.

Figure 5 shows the amount of network bandwidth consumed by PB for ICP and Cache Digests. For ICP, we have calculated the amount of traffic which would exist in the absence of Cache Digests. Additionally, we calculate only ICP traffic from the NLANR caches, and do not include ICP messages from other caches. In making this calculation, we lose some accuracy. For instance, Figure 5 and Table 3 do not show that PB receives slightly more ICP bytes than it sends[11]. Note that for cache digests, PB sends more than it receives. This is because the SD cache is significantly smaller, and there are a small number of child caches which take PB's digests, but PB does not take theirs.

Figure 6 shows the theoretical break-even points for Cache Digests vs. ICP. This can be used to determine the digest refresh period which will result in a lower bandwidth utilization. Our derivation follows.

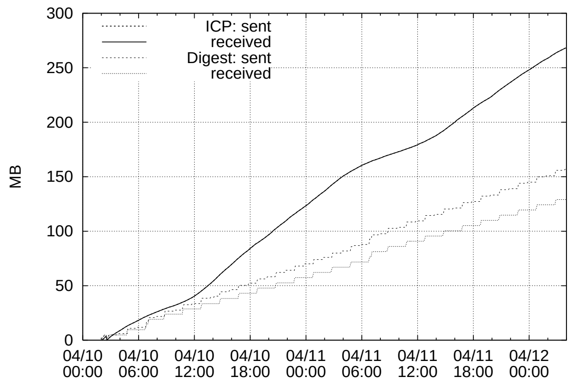
Given a cache size S, an average **per-peer** ICP query rate of R per second, and a digest refresh period of T, we draw the lines where the average traffic rates for Cache Digests and ICP are equal.

Squid uses 6 bits in the bloom filter for every object. For a given cache size S in GB and an average object size l in KB, the size of Squid's cache digest in MB is:

[11] This is because PB receives more queries than replies, and the ICP_QUERY message includes four extra bytes which are not present in replies.

Proxy	Message rate, msgs/hour			
	ICP		Digest	
	Sent	Received	Sent	Received
PB	75772	75772	7.1	3.8
UC	99725	99725	7.2	3.0
BO1	75383	75383	7.2	3.5
BO2	86596	86596	10.3	6.0
SV	16073	16073	7.9	2.9
SD	101895	101895	8.5	4.2

Table 4: Number of messages transmitted per hour for ICP and Cache Digests. Of course, Cache Digests require significantly fewer messages. Cache Digests use about one message per digest update. For this experiment, digests were expired with one hour intervals. Thus the number of digests received is approximately the number of cache peers that support Cache Digests. We send more digest messages than we receive because some messages go to peers which are either dead or do not have Cache Digest support.



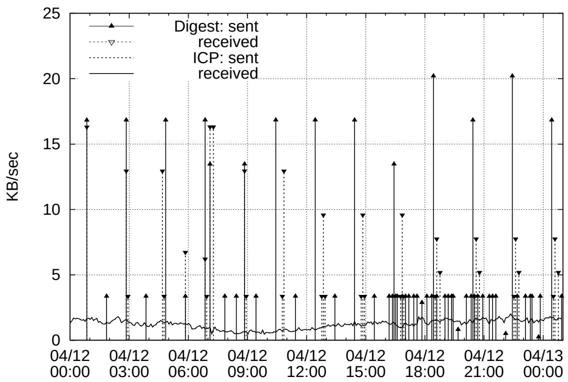


Figure 5: On the left, cumulative network traffic for Cache Digests and ICP. Here we have eliminated ICP traffic from non-NLANR caches. On the right we show the rate of traffic for both methods. These are five minute samples. Cache Digest traffic is very bursty while ICP traffic, by comparison, is very smooth. More "dense" Cache Digest traffic after *15:00* is the result of the improvement to digest update algorithm. Prior to that time, a cache would request digests from its peers almost at the same time. The graph clearly shows times when one, two, three, or more digests were sent or received with little or no gap (levels of line markers correspond to the number of digests transmitted during that 5 minute period). This unwanted synchronization led to large peaks in Cache Digests traffic rate. Peaks start to dissolve after we introduced an lower bound on the frequency of digest updates.

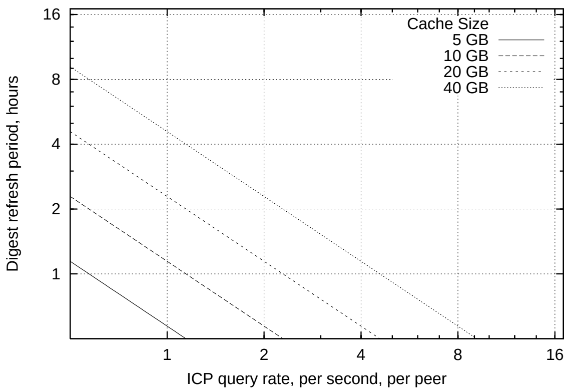


Figure 6: Theoretical break-even points for Cache Digests vs. ICP. The four lines represent different cache sizes. Larger cache sizes mean larger cache digest messages. For a given cache size, and a given ICP query rate (per peer), you can estimate the digest refresh period which will result in lower overall bandwidth utilization. Points above the lines represent lower bandwidth for Cache Digests, while points below the lines represent lower bandwidth for ICP. For example, a 20 GB cache receiving an average of 2 ICP queries per second from each of its peers will benefit from switching to cache digests with an update period of 1.15 hours or more.

$$S_{cd} = 0.75 \cdot \frac{S}{l}$$

The Cache Digest traffic rate, in MB/s, is given as:

$$\frac{S_{cd}}{3600T}$$

The ICP traffic rate, assuming an average ICP message size of 70 bytes, and R queries per second, is given (in bytes/sec) as:

$$2 \cdot 70 \; bytes \cdot R$$

Equating these two, and solving for T as a function of R,

$$T = \frac{S_{cd}}{0.504R}$$

Or in terms of the cache size S (GB) with an average object size l of 13 KB,

$$T = \frac{S}{8.74R}$$

6.7 Memory Usage

The size of a digest depends on the cache size. For experiments presented in this paper, Squid used 6 bits per object assuming full cache capacity. Thus, a 16 GB cache had a digest of approximately 1 MB. For less-than-full caches, digest size can be calculated based on the actual amount of disk space used with some "fudge factor" to allow for cache size growth.

The total memory requirement for running Cache Digest is proportional to the number of peers with digests. A cache peering with 4 neighbors required about 5-6 MB of RAM to store its own digest and digests of its peers. After all digests has been fetched, memory usage is stable.

7 Related Work

The Cache Array Routing Protocol (CARP) has been designed by the University of Pennsylvania and Microsoft[8]. With this scheme, requests are deterministically forwarded to a set of neighbor caches. The algorithm always forwards a request for the same URL to the same neighbor. CARP works well for Intranet hierarchies, but less well for loosely coupled, non-autonomous Internet cache peerings.

Povey and Harrison have proposed a Distributed Internet Cache[9] which is primarily targeted at eliminating upper level bottlenecks and scaling issues. In their scheme, upper level Web caches are replaced by "directory servers." Instead of serving objects via HTTP, these servers receive addition and deletion advertisements from lower level caches. Thus, an upper level server knows the contents of each lower level cache and may redirect one cache to another for a given object. The authors propose simple modifications to ICP for the exchange of advertisement messages.

Gadde, Rabinovich, and Chase have developed a similar system labeled CRISP[10]. Here, a central mapping service ties together some number of caches. The mapping service directs proxies to fetch cached objects from each other. The CRISP system works well for autonomously managed caches. For a wide-area configuration, the network RTT between the cache and a CRISP server may introduce significant delays.

Pei Cao and students at the University of Wisconsin, Madison have developed an object location technique based on Bloom Filters called "Summary Cache"[4]. Summary Cache extends ICP to allow "pushing" of Bloom Filters from parent caches to their children. Updates are supported via ICP as well. Summary Cache maintains a special table to track deletions from a Bloom Filter. The size of that table is 4 times the size of a local Bloom Filter. The table allows them to notify peers when objects are purged from the cache. The researchers tested their ideas using a log-driven simulation and small scale benchmarking with an instrumented version of Squid.

Cache Digest was designed and developed independently from Pei Cao's group. We regret having no knowledge of the Summary Cache project until it was completed leaving us no opportunity to influence the development. While the design objectives are probably the same, our understanding of practical matters seems to differ somewhat. We believe that push technology is not well suited for Squid and cooperative caching in general. With push caching, a parent proxy has to maintain state for all of its children, and there is no guarantee that the children are willing to accept fresh Bloom Filters as often as (and when) the parent proxy wants to generate them. Summary cache does not piggyback update messages and requires a special ICP message to "push" updates. We are also not sure that supporting deletions from a Bloom Filter is worth the overhead because most purged objects are old and are never requested again anyway.

Cache Digests require a common set of hash functions for lookups, but nothing else. Digest creators are free to choose the algorithm which builds the bloom filter (e.g., giving priority to some objects). Digest consumers need not know the details of such an algorithm. For Summary Cache, creators precisely specify the full algorithm details, and consumers must be able to recognize the algorithm, or the bloom filter is useless.

The Hypertext Caching Protocol[6] is a recently proposed replacement for ICP. As with ICP, HTCP is a UDP-based query/reply protocol. The primary difference is that an HTCP query includes full HTTP request headers, and HTCP replies include full HTTP reply headers. This addresses ICP's inability to accurately predict cache hits due to a lack of HTTP request headers such as *Cache-Control: Max-age*.

8 Acknowledgments

We are thankful to Henny Bekker (SURFNet, the Netherlands), Dancer (Australia), and other beta testers of Cache Digests.

This work was supported by grants of the National Science Foundation (NCR-9521745, NCR-9616602).

9 Conclusions and Future Work

In this study, we have shown that a *Cache Digest*, based on Bloom Filters, provides an object location function similar to ICP, but without any per-request delays. Median service times on our caches improved by 100 milliseconds or more when Cache Digests were used instead of ICP. Additionally, if the Cache Digest parameters (especially update frequency) are carefully selected, this technique consumes less network bandwidth overall. Cache Digests have increased memory requirements, approximately 1 MB per digest in this study.

Whereas ICP generates a steady stream of small packets, Cache Digest transfers occur in high-volume bursts (e.g. 1 MB every 2 hours). For this reason, Cache Digests may not be suitable for use over low-speed or highly congested network paths. This remains an area for future investigation, and we hope to receive good feedback regarding this from the user community.

The most significant drawback to Cache Digests is the number of false hits—as high as 18% for our caches. We can not realistically expect digests to be used for sibling relationships until this number is reduced to at least 5%. Therefore, our top priority task is to identify the primary reason for relatively high false hit ratio.

Related to this, we must also modify Squid to properly handle 504 (Gateway Timeout) replies for `only-if-cached` requests. Even with a 5% false hit ratio, we can not use digests for sibling relationships without `only-if-cached` support.

Our short study seems to support the idea that Cache Digests may not need to be kept fully up-to-date. With an update period of one hour (and no incremental updates), we witnessed essentially no change in hit/miss and true/false ratios. If other investigations reveal that incremental updates are beneficial, we intend to implement this functionality with customized HTTP request and reply headers.

Meanwhile, we will continue to tune our Cache Digest implementation, try different configurations, and gather additional performance data from the NLANR caches.

References

[1] D. Wessels and K. Claffy, "Internet cache protocol (ICP), version 2," *Network Working Group RFC 2186*, September 1997. `http://ds.internic.net/rfc/rfc2186.txt`.

[2] D. Wessels and K. Claffy, "Application of internet cache protocol (ICP), version 2," *Network Working Group RFC 2186*, September 1997. `http://ds.internic.net/rfc/rfc2187.txt`.

[3] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, July 1970.

[4] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," Tech. Rep. 1361, Department of Computer Science, University of Wisconsin-Madison, February 1998. `http://www.cs.wisc.edu/~cao/papers/summarycache.html`.

[5] R. Fielding *et al.*, "Hypertext transport protocol – HTTP/1.1," *Network Working Group RFC 2068*, January 1997. `http://ds.internic.net/rfc/rfc2068.txt`.

[6] P. A. Vixie, "Hyper text caching protocol (htcp/0.0)," *draft-vixie-htcp-proto-00.txt*, March 1998. `http://ds.internic.net/internet-drafts/draft-vixie-htcp-proto-00.txt`.

[7] D. Wessels, K. Claffy, and H.-W. Braun, "NLANR prototype web caching system." Research project funded by the National Science Foundation. `http://ircache.nlanr.net/`.

[8] V. Valloppillil and K. W. Ross, "Cache array routing protocol v1.0," *draft-vinod-carp-v1-03.txt*, February 1998. `http://ds.internic.net/internet-drafts/draft-vinod-carp-v1-03.txt`.

[9] D. Povey and J. Harrison, "A distributed internet cache," in *Proceedings of the 20th Australasian Computer Science Conference (to appear)*, February 1997. `http://www.psy.uq.edu.au:8080/~dean/project/`.

[10] S. Gadde, M. Rabinovich, and J. Chase, "Reduce, reuse, recycle: An approach to building large internet caches," in *Workshop on Hot Topics in Operating Systems (HotOS)*, April 1997. `http://www.cs.duke.edu/ari/cisi/crisp-recycle/`.

[11] D. Wessels, "Squid internet object cache." `http://squid.nlanr.net/`.